Verifying Anti-Security Policies Learnt from Android Malware Families

Verifying Anti-Security Policies Learnt from Android Malware Families

(Extended Abstract)

Wei Chen[1], Charles Sutton[1], David Aspinall[1], Andrew D. Gordon[1,2], Qi Shen[3], and Igor Muttik[4]

[1]University of Edinburgh, UK
[2]Microsoft Research Cambridge, UK
[3]Peking University, China
[4]Intel Security, UK

Android malware has been increasingly identified and organised into families [28, 36], e.g., Geinimi, Basebridge, Spitmo, Zitmo, and Ginmaster, etc. This human-decided organisation was based on some unexpected behaviours exhibited in malware instances, e.g., intercept incoming messages then send them out via Internet connections, load classes from a hidden payload then execute commands from remote servers, and send premium messages constantly, and so on; and malware instances in one family share some common unexpected behaviours [17, 29]. We study the problem of verifying Android applications to deny these behaviours. That is, (a) to formalise and learn unexpected behaviours from malware instances exploiting their family information, so-called anti-security policies; (b) and verify target applications against these policies efficiently, so as to decide whether a target application has any unexpected behaviour. Our main contributions are : (a) we implement a static analysis tool to construct an extended Büchi automaton for each Android application to approximate its behaviours, considering a broad range of features of Java and the Android framework [3]; (b) we develop an efficient machine-learning-centred method to construct sub-automata as anti-security policies from thousands of malware instances across hundreds of malware families; (c) we demonstrate the effectiveness of anti-security policy verification by showing how it helps reveal covert channels in a scenario of collusion attacks. We show that using the verification results against anti-security policies as input features, the classification performance on new malware detections is improved dramatically, in particular, the precision and recall are respectively 8% and 51% better than those using APIs calls and permissions as input features. We compare anti-security policies for malware families to their manual descriptions, which have been produced by malware analysts or third-party researchers [1, 2, 4, 5, 23], and demonstrate they compare well to these descriptions. This research has several potential benefits, including: help people get better understanding of potential threats hidden in mobile applications; provide hints for malware analysts before more expensive investigation; support automatic generation of malware analysis reports; and provide clear and friendly references for security policy designers, etc.

Formalisation. We characterise an application's behaviour by an extended Büchi automaton, i.e., finite and infinite control-sequences of events, actions, and annotated API calls, a so-called behaviour automaton. For example, the automaton $\mathcal{M}$ in Figure 1 tells us: an application will send messages after a user's action, e.g., click a button, touch the screen, or long-press a picture, etc., which is denoted by "click"; it tries to access Internet or send messages out when a short message is received; it also collects your device ID and phone number. All states in this automaton are accepting states because any prefix of an application's behaviour is its behaviour as well. We have designed and implemented

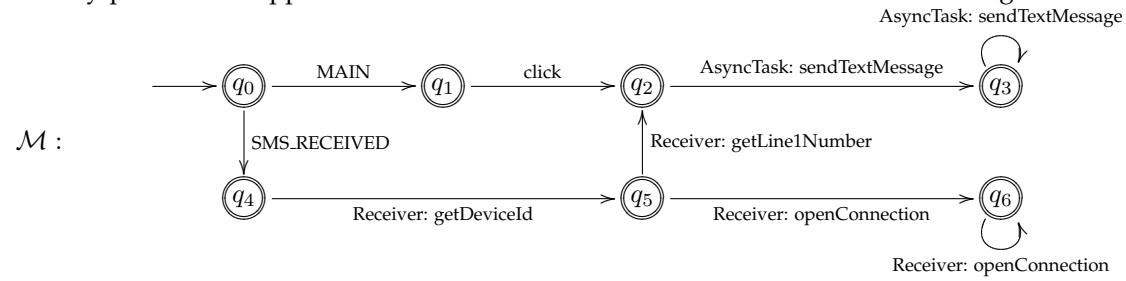

Figure 1: Behaviour Automaton of an Android Application

a static analysis tool to construct such a behaviour automaton from the assembly code of each Android application to approximate its behaviours. The Android platform tools `aapt` and `dexdump` are used to decompile these applications. To develop such a tool, we have to consider a broad range of features of Java and the Android framework, e.g., multi-threads, multi-entries, inter-procedural calls, callbacks, component life-cycle, inter-component communications, and runtime-registered listeners, etc. The choice of which features to include is a trade-off between efficiency and precision. Automata are much more accurate than the manifest information, e.g., permissions and actions, which can be extracted from the `AndroidManifest.html` file of an Android application. Compared with API calls appearing in code, the extended Büchi automata can capture more sophisticated behaviours. This is needed in practice, because: API calls appearing in code contain "noise" caused by dead code and libraries; and, some unexpected behaviours only arise when some API methods are called in certain orders. Also, some complex behaviours, e.g., the life-cycle of Android activity components, can only be modelled by using infinite sequences. On the other hand, automata are less accurate than data-flows. However, it is much easier to generate behaviour automata using our tool for applications en masse than

generating data-flows using tools like FlowDroid [10] or Amandroid [32]. In particular, people can annotate appealing API methods to generate behaviour automata more efficiently, rather than considering all data-dependence between statements. In our implementation, we use an extension of permission-governed API methods generated by PScout [11] as annotations.

Learning. An anti-security policy is a common abstract behaviour shared by applications in the same malware family. For example, let us consider a malware family called Ggtracker. A brief description of this family, which was

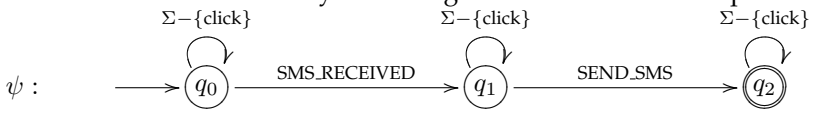

Figure 2: An Anti-Security Policy Learnt from a Malware Family

produced by Symantec [5], is as follows: *Android.Ggtracker is a Trojan horse for Android devices that sends SMS messages to a premium-rate number. It may also steal information from the device*. One of anti-security policies we construct from malware instances in this family is displayed as Figure 2. This policy ψ describes an unexpected behaviour of this family: send messages without any user's interaction after an incoming message is received. We outline our method to learn anti-security policies as follows. (a) *Abstract Behaviour Automata.* Since a behaviour might relate to several API calls, e.g., sendTextMessage, sendDataMessage, and sendMultipartTextMessage are all related to the behaviour of sending messages, to remove this kind of redundancy, we replace API names in a behaviour automaton by permissions or permission-like phrases without changing any node or transition; this we call the abstract behaviour automaton. For instance, the abstract behaviour automaton $\mathcal{A}$ of the behaviour automaton $\mathcal{M}$ is given in Figure 3, in which permission-governed API methods, e.g., sendTextMessage, getDeviceId, and openConnection, etc., are respectively replaced by their permissions. For some API methods which are not permission-governed but security-related, e.g., loadClass, loadLibrary, and loadUrl, etc., we replace them by pre-defined phrases. (b) *Feature Construction.* Once an abstract behaviour

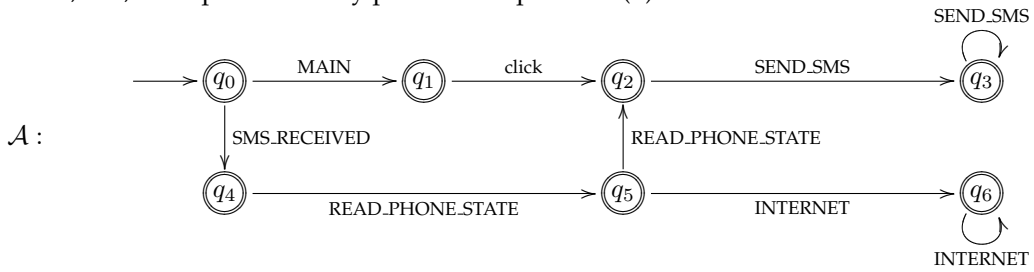

Figure 3: Abstract Behaviour Automaton of an Android Application

automaton has been constructed for each malware instance, we want to figure out: for each family, which behaviour is unexpected and which is normal. But, before any interesting pattern exploration, we have to decide which part of an application's behaviour is exclusive to itself and which part is shared with other applications; this we call feature construction. The space of features, which consists of intersection and difference between abstract behaviour automata, in theory, is exponential in the number of sample applications. So, we approximate it by searching for a salient subspace, e.g., $\{\mathcal{A}_j \in \{\oplus_{\mathcal{A}\in G}\mathcal{A} \mid \oplus \in \{-, \cap\}\} \mid \mathbf{w}_j \neq 0\}$, which is guided by behavioural difference between malware and benign applications. Here, $\mathbf{w}_j$ is the weight assigned to a feature, and a feature is salient if its weight is non-zero. These weights are assigned by a linear classifier, which is trained from a group G of malware instances mixed with an equal number of randomly-chosen benign applications, e.g., using L1-Regularized Logistic Regression [18, 30]. (c) *Learning Unexpected Behaviour.* An unexpected behaviour is a common behaviour shared by malware instances but rarely seen in benign applications. If we train a classifier, intuitively, a feature with a negative weight more likely indicates an unexpected behaviour while a feature with a positive weight more likely indicates a normal behaviour. This observation leads us to capture unexpected behaviour using features with negative weights. Further, for each malware family, we want to choose a subset of features so that it largely covers and is strongly associated with malware instances in this family. Formally, let us write $Pr(f|X)$ to denote the probability of a malware instance belonging to a family f if this instance has all features from X and $Pr(X|f)$ to denote the probability of having all features from X if a malware instance belongs to a family f. We adopt F_β-measure of them as the evaluation function to search for such subsets. A so-

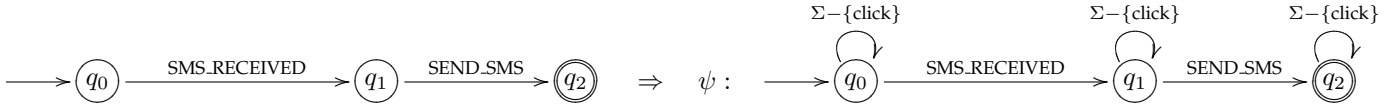

Figure 4: Construct an Anti-Security Policy from a Learnt Unexpected Behaviour

constructed unexpected behaviour of the Ggtracker malware family, as an example, is displayed as the left-hand side of Figure 4. (d) *Anti-Security Policies.* Many unexpected behaviours identified in target applications will not be the same as unexpected behaviours learnt from sample malware instances, but they might contain learnt unexpected behaviours as subsequences. For example, although the unexpected sequence SMS_RECEIVED.SEND_SMS is not accepted by the abstract behaviour automaton $\mathcal{A}$, this automaton does accept sequences containing SMS_RECEIVED.SEND_SMS as a subsequence, i.e., SMS_RECEIVED. READ_PHONE_STATE.READ_PHONE_STATE. SEND_SMS.SEND_SMS$^{\leqslant\omega}$. Here, the expression SEND_SMS.SEND_SMS$^{\leqslant\omega}$ denotes that the application will send text messages several times or infinitely often. So, if a behaviour contains a learnt unexpected behaviour as a subsequence, we consider this behaviour as unexpected as well. We call this generalisation of learnt unexpected behaviours *anti-security policies*, in particular, we

construct them directly from learnt unexpected behaviours by adding an edge labelled with $\Sigma - \{\text{click}\}$ to each state of each automaton, shown as the right-hand side of Figure 4. Here, we use Σ to denote the collection of events, actions, and permission-like phrases.

Verification. Once anti-security policies $\psi \in \mathcal{P}$ are constructed from sample malware instances across families, we want to check whether the abstract behaviour automaton $\mathcal{A}$ of a target application satisfies the negation of an anti-security policy, i.e., $\forall \psi \in \mathcal{P} \,.\, \mathcal{A} \models \neg\psi$. Equally, if there exists a $\psi \in \mathcal{P}$ such that $\mathcal{A} \cap \psi \neq \emptyset$, then we consider there is a security fault in this application with respect to $\mathcal{P}$. For instance, since the intersection between an application's abstract behaviour automaton given in Figure 3 and the anti-security policy given in Figure 2 is not empty, we consider this application is unsafe with respect to $\{\psi\}$. To demonstrate the effectiveness of the anti-policy verification, we present a scenario of collusion attacks exploiting channels created by inter-component-communications and show how to apply the above verification method to help identify potential threats. There are three roles in this scenario: an agent, a postman, and an invader. An agent is an application which has permissions to collect information and a postman is an application with capability of sending information out. They are normally benign applications, e.g., message managers, email clients, and social network applications, etc. An invader will explore service supplied by agents and postmen, to steal information without necessity of declaring any permission. As a proof-of-concept we implement some experimental programs and display their behaviour automata in Figure 5. In this implementation, the invader will

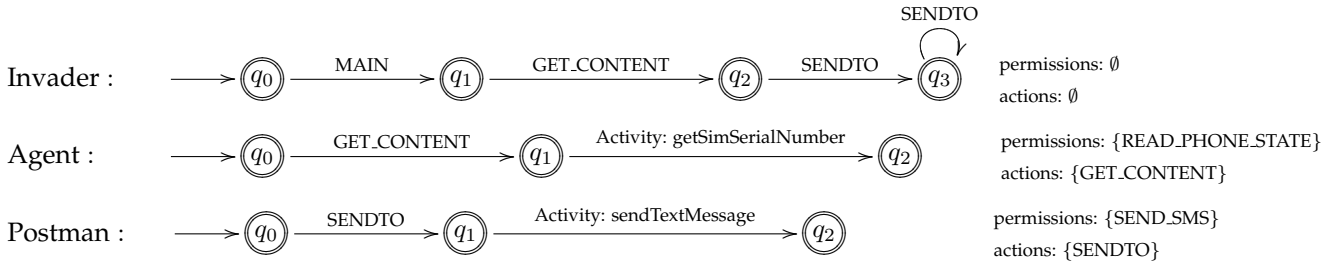

Figure 5: An Implementation of the Agent-Invader-Postman Scenario

make use of service supplied by the agent and the postman to collect user's IMSI number then send it out via SMS. First, it invokes the API startActivityForResult with a GET_CONTENT intent in the callback onCreate of an activity; then it will invoke the API startActivity with a SENDTO intent in the callback onActivityResult of the same activity, when the IMSI number is received. Since the invader does not require any permission or action, and except startActivity and startActivityForResult, it does not invoke any other API method, only considering the invader's behaviour, there is no suspicious. However, if we combine behaviour automata through replacing an action by how an application deals with this action, i.e., GET_CONTENT by Activity:getSimSerialNumber and SENDTO by Activity:sendTextMessage, we will get an abstract behaviour automaton displayed in Figure 6. Given an anti-security policy ϕ, the intersection between

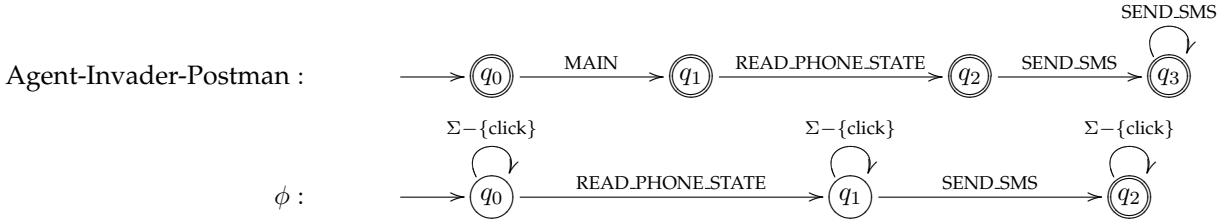

Figure 6: The Combined Abstract Behaviour Automaton of the Agent-Invader-Postman Scenario

these two automata is not empty. So, an Android environment containing these three applications does have a potential security fault with respect to the anti-security policies $\{\phi\}$.

Evaluation. We want to demonstrate that learnt unexpected behaviours can capture most unexpected behaviour exhibited in sample malware instances. We collected around 4,000 identified malware instances from different sources [1, 23, 28]. These malware instances have been manually investigated and organised into around 200 families, e.g., Basebridge, Spitmo, and Zitmo, etc., by malware analysts or third-party researchers. Some malware families contain hundreds of instances, e.g., Geinimi, Ginmaster, and Droiddream; some families only contain several instances, e.g., Arspam and Ggtracker. We collected these human-authored descriptions from online malware analysis reports [1, 2, 4, 5, 23]. By applying the formalisation and learning approaches we have discussed, for each family unexpected behaviours are constructed from its malware instances. We list manual descriptions and learnt unexpected behaviours of several prevalent families in Table 1. Unexpected behaviours are specified in regular expressions and ω-languages. We use the Greek letter ϵ to denote the empty string. The expression $\_^{\leqslant\omega}$ means $\_^*|\_^\omega$ and the expression $\_^{\leqslant\omega^+}$ denotes $\_^+|\_^\omega$. The concatenation is extended as usual, i.e., $(\_^\omega).\alpha = \_^\omega$ with $\alpha \in \Sigma^{\leqslant\omega}$. A subjective comparison shows that learnt unexpected behaviours are comparable to these manual descriptions. Learnt unexpected behaviours also reveal trigger conditions of some behaviours, e.g., the expression BOOT_COMPLETED.SEND_SMS denotes that after the device finishes booting this application will send a message out, the expression UMS_CONNECTED.LOAD_CLASS means that when a USB massive storage is connected to the device this application will load some code from a library or a payload, and the unexpected behaviour for Droiddream shows that if the phone state changes this application will collect information then access Internet, etc. Within manual descriptions displayed in Table 1, only two behaviours are not captured by learnt unexpected behaviours, i.e., gain root access for Droiddream and the behaviour of Spitmo. Some anti-security

Family	*Manual Description*	*Learnt Unexpected Behaviour in Regular Expressions and ω-Languages* *The letter ϵ denotes the empty string, the expression $\_^{\leqslant\omega}$ means $\_^*\mid\_^\omega$, and $\_^{\leqslant\omega^+}$ denotes $\_^+\mid\_^\omega$.*
Arspam	Sends spam SMS messages to contacts on the compromised device [5].	1. BOOT_COMPLETED . SEND_SMS
Anserverbot	Downloads, installs, and executes payloads [1].	1. UMS_CONNECTED . LOAD_CLASS$^{\leqslant\omega}$. (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET) . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET | LOAD_CLASS)$^{\leqslant\omega}$
Basebridge	Forwards confidential details (SMS, IMSI, IMEI) to a remote server [2]. Downloads and installs payloads [1, 5].	1. UMS_CONNECTED . (INTERNET | LOAD_CLASS | READ_PHONE_STATE | ACCESS_NETWORK_STATE)$^{\leqslant\omega^+}$
Cosha	Monitors and sends certain information to a remote location [5].	1. MAIN . click . (click | ACCESS_FINE_LOCATION | DIAL)$^{\leqslant\omega}$. DIAL . (click | ACCESS_FINE_LOCATION | DIAL)$^{\leqslant\omega}$. (INTERNET | ϵ) 2. SMS_RECEIVED . (INTERNET | ACCESS_FINE_LOCATION)$^{\leqslant\omega^+}$
Droiddream	Gains root access, gathers information (device ID, IMEI, IMSI) from an infected mobile phone and connects to several URLs in order to upload this data [1, 2].	1. PHONE_STATE . (ACCESS_NETWORK_STATE | READ_PHONE_STATE$^{\leqslant\omega^+}$. INTERNET) . (ACCESS_NETWORK_STATE | INTERNET)$^{\leqslant\omega}$
Fakelogo	Sends SMS messages to premium rate numbers [4].	1. BOOT_COMPLETED . RUN$^{\leqslant\omega^+}$ 2. BOOT_COMPLETED . READ_PHONE_STATE$^{\leqslant\omega^+}$ 3. MAIN . click . SEND_SMS . (SEND_SMS | ϵ) 4. MAIN . SEND_SMS
Geinimi	Monitors and sends certain information to a remote location [5].	1. ϵ | MAIN . click$^{\leqslant\omega^+}$. VIBRATE . (click | VIBRATE)$^{\leqslant\omega}$. RESTART_PACKAGES . (MAIN . (click | VIBRATE)$^{\leqslant\omega}$. RESTART_PACKAGES)$^{\leqslant\omega}$ 2. BOOT_COMPLETED . (ACCESS_NETWORK_STATE | click | INTERNET | RESTART_PACKAGES | ACCESS_FINE_LOCATION)$^{\leqslant\omega^+}$
Ggtracker	Monitors received SMS messages and intercepts SMS messages [2]	1. MAIN . READ_PHONE_STATE 2. SMS_RECEIVED . SEND_SMS
Ginmaster	Sends received SMS messages to a remote server [23]. Downloads and installs applications without user concern [23].	1. BOOT_COMPLETED . LOAD_CLASS 2. MAIN . SEND_SMS
Spitmo	Filters SMS messages to steal banking confirmation codes [5].	1. NEW_OUTGOING_CALL . READ_PHONE_STATE . INTERNET . (INTERNET | ϵ)
Zitmo	Opens a backdoor that allows a remote attacker to steal information from SMS messages received on the compromised device [5].	1. SMS_RECEIVED . SEND_SMS 2. MAIN . READ_PHONE_STATE 3. MAIN . SEND_SMS

Table 1: Learnt Unexpected Behaviour versus Manual Description of Malware Families

policies in LTL [24] which are constructed from these unexpected behaviours are as follows:

G ($\neg$click $\wedge$ (SMS_RECEIVED $\rightarrow$ **F** SEND_SMS)) (from Ggtracker and Zitmo);

G ($\neg$click $\wedge$ (UMS_CONNECTED $\rightarrow$ **F** (LOAD_CLASS $\vee$ INTERNET $\vee$ READ_PHONE_STATE))) (from Basebridge);

G ($\neg$click $\wedge$ (BOOT_COMPLETED $\rightarrow$ **F** SEND_SMS)) (from Arspam);

G ($\neg$click $\wedge$ (BOOT_COMPLETED $\rightarrow$ **F** RESTART_PACKAGES)) (from Geinimi);

G ($\neg$click $\wedge$ (PHONE_STATE $\rightarrow$ **F** (READ_PHONE_STATE $\rightarrow$ **F** INTERNET))) (from Droiddream).

They reveal the main unexpected behaviours exhibited in malware instances of these families: (a) intercept incoming messages (and calls) and send them out to remote servers; (b) (download and) run code in hidden payloads (then execute commands from C&C servers); (c) collect personal information, e.g., locations, IMEI, IMSI, and MAC addresses, and so on; (d) send (premium) SMS messages.

We also want to show that verifying anti-security policies helps improve the robustness of malware classifications. We collected $3,000$ malware instances which have been discovered before 2014 and $3,000$ randomly-chosen benign applications. They include all malware instances from the Malware Genome Project [1, 36] and most malware instances from the Mobile-Sandbox [9, 27]. These malware instances have been manually investigated and organised into around 200 families by third-party researchers [1, 27, 36] and malware analysts [2, 4, 5, 23]. By reading online malware analysis reports [1, 2, 4, 5, 23] of these families, we understand what bad things happen in these malware instances. We divided them into the training set and the validation set. Each of them includes $1,500$ malware instances across all families and $1,500$ benign applications. We collected $1,500$ malware instances which were discovered in 2014 and randomly chose $1,500$ benign applications, to form the testing set. These malware instances were from the Intel Security and have been investigated by malware analysts, but there is no family information, i.e., we have no idea of unexpected behaviours of these malware instances. We extracted permissions and API calls from these applications as input features. We constructed behaviour automata for these applications then applied the method discussed earlier to learn anti-security policies from malware instances in the training set. We applied the verification method discussed earlier to check whether an application satisfies the negation of an anti-security policy. We collected these verification results as input features as well. We adopt the L1-Regularized Logistic Regression [20, 30] as the training method, then train and test classifiers using different combinations of features. The classification performance is reported in Table 2. The singletons denote labels extracted from the edges of abstract behaviour automata and a pair (f, g) denotes that $f.g$ is a subsequence of a sequence which is accepted by the automaton of an anti-security policy. It confirms: (a) Anti-security policies dramatically improve the classification performance on new malware instances. The classification performance using API calls and permissions as input features is very good on the validation set, i.e., the precision and recall are respectively 93% and 98%. However, this is just an over-fitting to the training set, since its performance on the testing set is bad, i.e., the precision is 65% and recall is 15%. This means that a lot of new unexpected behaviours cannot be captured by API calls and permissions. By using the verification results against anti-security policies as input features, we improve the precision to 73% and the recall to 66%. (b) The generalisation from unexpected behaviours to anti-security policies helps improve the classification performance. (c) The family information helps improve the

feature	*validation set*				*testing set*				#salient/#feature
	precision	recall	FPR	FNR	precision	recall	FPR	FNR	
syntax-based features									
permissions	89%	99%	17%	0.8%	53%	21%	19%	79%	59/175
apis	91%	98%	12%	2%	61%	15%	9%	85%	1443/52432
apis & permissions	**93%**	**98%**	10%	2%	**65%**	**15%**	8%	85%	735/52607
semantics-based features									
unexpected behaviours	66%	91%	64%	9%	53%	74%	65%	26%	634/886
singletons	73%	90%	45%	10%	64%	72%	40%	28%	85/87
pairs	68%	92%	59%	8%	55%	69%	57%	31%	344/7568
policies	75%	87%	10%	13%	69%	66%	30%	34%	581/886
singletons & policies	76%	90%	37%	10%	70%	67%	28%	33%	963/973
policies for families	**72%**	**72%**	38%	28%	**73%**	**66%**	25%	34%	**131/131**
singletons & policies for families	76%	88%	38%	12%	70%	67%	29%	33%	214/218
mixed features									
all	95%	99.5%	7.7%	0.5%	65%	7.5%	4%	92.4%	870/61149

Table 2: Classification performance using different features. (FPR—False Positive Ratio. FNR—False Negative Ratio.)

classification performance on new malware detections, not only the usual mesaures, e.g., precision and recall, but also the minimum number of features which are actually used in a linear classifier, i.e., totally 131 features in policies for families were used, rather than 581 features in policies.

Conclusion. Machine learning methods have been applied in Android malware detection [6, 12, 15, 17, 19, 35], clustering [29] and simple explanation [9] of malicious behaviour. However, learning simple, understandable, and verifiable properties from identified malware instances is more challenging and has not yet been considered. To the best of our knowledge, our approach is the first one to automatically construct anti-security policies from Android malware instances exploiting their family information. We show that these learnt anti-security policies can be used for verification, to improve the classification performance on new malware detections, and to help people's understanding of malicious behaviours in these families. The idea of characterising applications' behaviours as automata is similar with the behaviour abstraction in [13, 33]. Abstract behaviour automata are close to permission-event graphs [16], embedded call graphs [19], and behaviour graphs [34]. But, none of them has been exploited to generate verifiable properties. Anti-security policies we have constructed can be considered as instances of security automata [25] and safety and liveness properties [7]. Our verification approach is the same as the automata-theoretic model checking [31]. More sophisticated approaches can be found in a recent study of LTL-model checking [26]. Some benign and malicious properties specified in LTL were verified against hundreds of Android applications in [16]. Also, totally 19 malicious properties for Android applications were manully constucted and specified as first-order LTL formulae in [21]. Unfortunately, these properties were manually composed. Anti-security policies which are automatically constructed from malware instances, compared with manually-composed properties, will be easier to be updated on the changes of unexpected behaviours exhibited in new malware families. Among others, Angluin's [8] and Biermann's [14] algorithms were developed to learn regular expressions from sample finite strings. To apply similar ideas in anti-security policy construction, we have to extend these algorithms to learn Büchi automata from infinite strings, which is an open problem [22], not to mention it is hard to extract enough finite and infinite strings from abstract behaviour automata. Since learnt anti-security policies are restricted to the family information of malware instances and many malware instances have no family information, in further work, we want to construct security as well as anti-security policies by exploring behavioural difference between benign applications and malware, especially when the family information is unavailable.

[1] Malware Genome Project. `http://www.csc.ncsu.edu/faculty/jiang/alerts.html`, 2012.

[2] Forensic Blog. `http://forensics.spreitzenbarth.de/android-malware/`, 2014.

[3] Android Developers. `http://developer.android.com/index.html`, 2015.

[4] Juniper Networks. https://www.juniper.net/security/auto/includes/mobile_signature_descriptions.html, 2015.

[5] Symantec Security Response. `http://www.symantec.com/security_response/`, 2015.

[6] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, pages 86–103. Springer, 2013.

[7] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[9] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. *Proc. of 17th Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269. ACM, 2014.

[11] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY,

USA, 2012. ACM.

[12] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2010.

[13] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification*, volume 6418 of *LNCS*, pages 168–182. Springer Berlin Heidelberg, 2010.

[14] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[15] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[16] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium*, 2013.

[17] L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14. ACM, 2014.

[18] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[19] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54. ACM, 2013.

[20] T. Haštie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, New York, 2001.

[21] J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Runtime Verification 2015*, Vienna, Austria, sep 2015.

[22] M. Leucker. Learning meets verification. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer Berlin Heidelberg, 2007.

[23] McAfee Threat Center. `http://www.mcafee.com/uk/threat-center.aspx`, 2015.

[24] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[25] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[26] F. Song and T. Touili. Ltl model-checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 416–431. Springer Berlin Heidelberg, 2013.

[27] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815, New York, NY, USA, 2013. ACM.

[28] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.

[29] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.

[30] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[31] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.

[32] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341. ACM, 2014.

[33] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, July 2002.

[34] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, September 2014.

[35] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new Android malware detection approach using bayesian classification. In *AINA*, pages 121–128. IEEE Computer Society, 2013.

[36] Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109. IEEE Computer Society, 2012.